
THE BASICS OF PERFORMANCE-MONITORING HARDWARE

PERFORMANCE-MONITORING FEATURES PROVIDE DATA THAT DESCRIBE HOW AN APPLICATION AND THE OPERATING SYSTEM ARE PERFORMING ON THE PROCESSOR. THIS INFORMATION CAN GUIDE EFFORTS TO IMPROVE PERFORMANCE.

••••• Most modern, high-performance processors have special, on-chip hardware that monitors processor performance. Data collected by this hardware provides performance information on applications, the operating system, and the processor. These data can guide performance improvement efforts by providing information that helps programmers tune the algorithms used by the applications and operating system, and the code sequences that implement those algorithms.

Performance event monitoring

Performance events can be grouped into five categories: program characterization, memory accesses, pipeline stalls, branch prediction, and resource utilization. Program characterization events help define the attributes of a program (and/or the operating system) that are largely independent of the processor's implementation. The most common examples of these events are the number and type of instructions (for example, loads, stores, floating point, branches, and so on) completed by the program. Memory access events often comprise the largest event category and aid performance analysis of the processor's memory hierarchy. For example, memory events can count references and misses to various caches and transactions on the processor memory bus. Pipeline stall event information helps users analyze how well the

program's instructions flow through the pipeline. Processors with deep pipelines rely heavily on branch prediction hardware to keep the pipeline filled with useful instructions. Branch prediction events let users analyze the performance of branch prediction hardware (for example, by providing counts of mispredicted branches). Resource utilization events let users monitor how often a processor uses certain resources (for example, the number of cycles spent using a floating-point divider).

Performance-monitoring hardware

Performance-monitoring hardware typically has two components: performance event detectors and event counters. By properly configuring the event detectors and counters, users can obtain counts of a variety of performance events under various conditions.

Users can configure performance event detectors to detect any one of several performance events (for example, cache misses or branch mispredictions). Often, event detectors have an event mask field that allows further qualification of the event. For example, the Intel Pentium III's event to count load accesses to the level 2 cache (L2_LD) has an event mask that lets event detectors monitor only accesses to cache lines in a specific state—modified, shared, exclusive, or invalid.¹

The event detector configuration also allows qualification by the processor's current

Brinkley Sprunt
Bucknell University

privilege mode. Operating systems use supervisor and user privilege modes to prevent applications from accessing and manipulating critical data structures and hardware that only the operating system should use directly. When the operating system is executing on the processor, the privilege mode is supervisor; when an application is executing on the processor, the privilege mode is user. As such, the ability to qualify event detection by the processor's privilege mode allows counting of events caused only by the operating system or only by an application. Configuring the event detector to detect events for both privilege modes counts all events.

In addition to counting events detected by the performance event detectors, users can configure performance event counters to count only under certain edge and threshold conditions. The edge detection feature is most often used for performance events that detect the presence or absence of certain conditions every cycle. For these events, an event count of one represents a condition's presence and zero indicates its absence. For example, a pipeline stall event indicates the presence or absence of a pipeline stall on each cycle. Counting the number of these events gives the number of cycles that the pipeline stalled.

However, the edge detection feature can also count the number of stalls (more specifically, the number of times a stall began) rather than just the total number of cycles stalled. With edge detect enabled, the performance counter will increment by one only when the previous number of performance events reported by the event detector is less than the current number being reported. So when the event detector reports zero events on a cycle followed by one event on the next cycle, the event counter has detected a rising edge and will increment by one. It's usually possible to invert the sense of the edge detection to count falling edges. For these events, disabling the edge detection feature counts stall durations, and enabling edge detection counts the number of stalls. Dividing the total stall duration by the number of stalls gives the average number of cycles stalled for a particular stall condition.

The event counter's second major feature is threshold support. This capability lets the event counter compare the value it reports each cycle to a threshold value. If the report-

ed value exceeds the threshold, the counter increments by one. The threshold feature is only useful for performance events that report values greater than one each cycle. For example, superscalar processors can complete more than one instruction per cycle. Selecting instructions completed as the performance event and setting the counter threshold to two would increment the counter by one whenever three or more instructions complete in one cycle. This provides a count of how many times three or more instructions completed per cycle.

Performance profiles

Although performance event detectors and counters can easily detect the presence of a performance problem and let the user estimate the severity of the problem, it's often necessary to find the locations of the code (whether in the application or the operating system) that are causing the performance problem. Knowing the source of the performance problem lets programmers alter the high-level algorithms used by the application and/or the low-level code to avoid or reduce the problem's impact. To illustrate how performance counters can help create a profile that identifies the major sources of performance problems, let's first review the goals and techniques used to create time-based profiles.

Time-based profiles

A common technique to identify areas upon which to focus tuning efforts is to obtain a time-based profile of the application. A time-based profile estimates the percentage of time an application spends in its major sections.² Focusing tuning efforts on the application's most frequently executed sections maximizes the benefits of any performance-tuning changes made to the code.

A time-based profile relies upon interrupting an application's execution at regular time intervals. During each interrupt, the interrupt service routine saves the value of the program counter. Once the application completes, the user can create a histogram that shows the number of samples collected for each program counter value. Assuming that the histogram draws from many program counter samples, it will show the application's most frequently executed sections.

Event-based profiles

A technique similar to that for creating a time-based profile can help collect an event-based profile. An event-based profile is a histogram that plots performance event counts as a function of code location. Instead of interrupting the application at regular time intervals (as is done to create a time-based profile), the performance-monitoring hardware interrupts the application after a specific number of performance events has occurred. So just as a time-based profile indicates the most frequently executed instructions, an event-based profile indicates the most frequently executed instructions that cause a given performance event.

To support event-based sampling (EBS), performance-monitoring hardware typically generates a performance monitor interrupt when a performance event counter overflows. To generate an interrupt after N performance events, the performance counter is initialized to a value of overflow minus N before being enabled. A performance monitor interrupt service routine (ISR) handles these interrupts. The ISR saves sample data from the program (for example, the program counter) and re-enables the performance event counter to cause another interrupt after N occurrences of a certain performance event. After the application finishes executing, the user can plot the data samples saved by the ISR to create an event-based profile.

Improving performance

To illustrate the use and benefits of performance-monitoring hardware, let's examine two examples. The first example demonstrates how performance-monitoring data help tune the memory accesses of a high-level algorithm for better performance on a specific processor. The second example demonstrates how performance-monitoring data can find a low-performing code sequence, which programmers can then modify to improve performance.

Identifying memory access problems

Sometimes, algorithms developed for a particular processor generation don't perform as well on a successive generation. Consider the following simple algorithm for finding prime numbers: The algorithm creates a large array where each element in the array represents an integer and begins by initializing all array ele-

ments to 0. To find prime numbers, the algorithm repeatedly steps through the array with successively greater strides starting with a stride of 2. On all but the initial steps, storing a value of 1 into the element marks the array elements. This mark indicates that the number corresponding to the array element is not prime. For example, the first pass through the array would use a stride of 2 and would mark locations 4, 6, 8, and so on. The second pass would use a stride of 3 and mark locations 6, 9, 12, and so on. Once the algorithm passes through the entire array, the array elements that are still 0 correspond to prime numbers.

Now, consider how this algorithm performs on the original Pentium and Pentium Pro processors.³ Both processors have an 8-Kbyte L1 data cache, but the Pentium Pro has an additional 256 Kbytes of unified L2 cache. In general, it seems that the overall performance of the Pentium Pro would surpass that of the Pentium because of its more advanced design and the presence of the large, unified L2 cache. However, a subtle difference in the allocation policies for these caches can cause some performance surprises. The Pentium does not allocate a new cache line for a store that misses the cache, but the Pentium Pro does. When the Pentium Pro allocates a new line to the caches, it must load the whole line—32 bytes of data—from memory.

In steady state, the prime number algorithm, described previously, accesses a large array (much larger than the caches) using a large stride. Thus, almost every store to the prime number array will miss the caches. On the Pentium, each store miss results in one bus transaction between the processor and memory because the Pentium does not allocate a new cache line. However, because the Pentium Pro allocates a new cache line on each store miss, each store miss causes four bus transactions to load the 32-byte cache line using the 8-byte-wide processor memory bus. Also note that, in steady state, most of the lines in the Pentium Pro caches are dirty (they have data that is more recent than that in main memory), because they are holding the 1s stored in the prime number array. As such, when the Pentium Pro allocates a new line, it must write a dirty line back to memory, requiring another four bus transactions. As such, the steady-state behavior of this algorithm causes only one

bus transaction for each store to the prime number array for the Pentium, but causes eight bus transactions for the Pentium Pro.³ This eight-times difference in required bus bandwidth is a bottleneck for the Pentium Pro, and its performance on this algorithm is low compared to that of the Pentium.

Initially, the reasons for the Pentium Pro's lower performance on the prime number algorithm were far from obvious. However, Intel's analysis of the cache and bus performance data collected using the hardware performance monitors made this problem apparent. Once the Intel computer architects understood the problem, they realized that a slight change in the algorithm would substantially reduce the processor memory traffic for the Pentium Pro. Instead of blindly storing a 1 in the array elements on each algorithm step, the algorithm should first test the array value. It should only write a 1 if the existing array value is 0. This simple, one-line change to the algorithm eliminates the majority of dirty cache lines because it only writes each element in the prime number array once. This change significantly reduces processor memory traffic for the Pentium Pro by eliminating the write back of dirty cache lines to main memory. It brings the Pentium Pro's performance to a level on par with or better than that of the Pentium. Without the performance-monitoring capabilities of these processors, the investigation and resolution of this performance problem would have been much more difficult.

Eliminating partial register stalls

For another example of the usefulness of hardware performance-monitoring features, consider the performance problem of partial register stalls on P6-based Pentium processors (these include the Pentium Pro, Pentium II, and Pentium III along with their Xeon and Celeron versions).⁴ On these processors, a partial register stall occurs when an instruction writes the lower 8 or 16 bits of a 32-bit register and another, closely following instruction reads the full 32 bits from the register. Here, register renaming⁵ causes each write of a register to go to a different physical register. Consequently, the newly written 8 or 16 bits reside in a different physical register than the unchanged upper bits. A subsequent instruction cannot read the register's full 32 bits until

the partial components of the register's value are merged as the instructions retire. This situation causes a stall of at least seven cycles, but typically lasts from 10 to 14 cycles.

To determine whether or not an application causes partial register stalls, a user can configure the performance event detectors and event counters to count the number of partial-register-stall cycles as the application runs. If the frequency of these cycles as a percentage of the application's total cycles is significant (that is, greater than 3 percent), finding and eliminating the partial register stalls will probably increase performance.

Once users identify partial register stalls as a significant performance problem, they can use the EBS support of the performance counters to locate the most frequent occurrences of these stalls. Users can use the collected samples to create a histogram of sample counts versus code location. This histogram identifies the specific locations in the code that cause the majority of partial register stalls so programmers can modify the code to eliminate them.

Alternative approaches

Several other approaches to collecting processor performance data don't rely on performance-monitoring hardware. These approaches typically require modifying the application to collect data about its own execution, as with the Etch tool.⁶ This instrumentation of the application requires either rebuilding it from source code or modifying its executable version. The modified version adds additional code to collect various data, which typically include instruction trace and memory reference data. Other tools then process these data to produce performance data. However, this approach is sometimes difficult to use—it's not always feasible to rebuild an application. The source code and build tools are not always available, for example. Also, these approaches can disturb the application's behavior, bringing into question the validity of the collected data.

Another way to collect processor performance data is by using a simulator to model the processor as it executes the application. This simulation approach can yield a wealth of data (for example, detailed data on pipeline stalls, branch prediction, cache performance, and so on). However, processor manufactur-

Software tools for performance-monitoring hardware

A variety of software tools provide a high-level interface and analysis capabilities for performance-monitoring hardware. The Performance Application Programming Interface (PAPI) tool suite (<http://icl.cs.utk.edu/projects/papi/>) provides a common interface to performance-monitoring hardware for many different processors, including Alpha, Athlon, Cray, Itanium, MIPS, Pentium, PowerPC, and UltraSparc. Intel's VTune Performance Analyzer (<http://developer.intel.com/software/products/vtune/vtune60/index.htm>) supports the performance-monitoring hardware of all Intel Pentium and Itanium processors, and provides additional performance analysis tools such as call graph profiling and processor-specific tuning advice. The Digital (now Hewlett-Packard) Continuous Profiling Infrastructure (<http://www.tru64unix.compaq.com/dcp/~/index.html>) for Alpha processors uses performance-monitoring hardware to provide a low-overhead facility for performance monitoring of an entire system. The Rabbit Performance Counter Library (<http://www.scl.ameslab.gov/Projects/Rabbit>) provides a high-level interface to P6-based Pentium processors and AMD Athlon processors on Linux systems. The Brink and Abyss tools (http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtml) provide a high-level interface to the Pentium 4 performance-monitoring features on a Linux system.

ers do not usually provide simulators for advanced processor designs. These simulators are also difficult to develop without detailed information about the processor's implementation, details of which the manufacturer does not typically divulge. Even if a simulator is available, its accuracy is always in question, and it's often difficult to get a complex application to run in a simulated environment. Simulation speed is also significantly slower than running the application on the real processor, slowing data collection and making it infeasible to simulate some applications.⁷

Using performance-monitoring hardware has several distinct advantages over these other approaches. Having the processor itself actually collect performance data as it executes an application avoids several problems. First, the application and operating system remain largely unmodified, apart from the addition of drivers in the operating system to enable access to the hardware performance counters. As such, using hardware performance counters does not require tools to instrument the application, and the disturbance to the system under analysis is minimal. Second, not using a simulation of the application, operating system, or processor ensures that the accuracy of the collected event counts. Third, performance-monitoring hardware collects data on the fly as the application executes, allowing full-speed data collection and avoiding the slowness of simulation-based approaches. Fourth, this approach can collect

data for both the application and the operating system.

These advantages often make hardware performance monitoring the preferred, and sometimes only, choice for collecting processor performance data.

Performance-monitoring capabilities today

Many of today's high-performance processors have hardware performance-monitoring capabilities. The "Software tools for performance-monitoring hardware" sidebar contains a discussion of software interface and analysis tools for some of these processors.

Advantages

Intel's P5-based processors (these include the original Pentium and Pentium MMX) and P6-based processors provide two performance counters that support privilege mode qualification, threshold comparisons, and interrupt generation on counter overflow.¹ These processors also provide a large set of performance events (more than 80 on the Pentium III). AMD's Athlon processor⁸ provides four performance counters with capabilities similar to the Pentium III, but with a smaller set of performance events (about 25). IBM's PowerPC 750 processor⁹ provides four counters with capabilities similar to the Pentium III and supports more than 40 events. Motorola's PowerPC 7450 provides six counters and over 200 events.¹⁰ An additional feature of the PowerPC performance-monitoring hardware is the ability to count performance events only for marked processes on a system running multiple processes. This feature lets a process indicate that its events should be counted, excluding those events generated by any unmarked process.

Intel's Itanium processor provides four performance counters and over 90 performance events.¹¹ In addition to supporting privilege-mode qualification, threshold comparisons, and interrupt generation on counter overflow, Itanium allows event qualification by opcode as well as instruction and/or data address range. Itanium events also account for all major single- and multicycle stall and flush conditions. These events let the user determine the percentage of all cycles spent on various activities, such as execution, data and instruction accesses, and branch mispredic-

tions. Unlike most other processors, Itanium contains a set of event address registers that records the instruction and data addresses for cache and transition look-aside buffer (TLB) misses. These event address registers identify data addresses and structures that cause frequent cache and TLB misses.

Sun's UltraSparc I and II provide two performance counters and qualification by privilege mode for over 20 events, but they have no threshold support and cannot generate interrupts on counter overflow.¹² Compaq's Alpha 21264 provides two performance counters that support privilege-mode qualification and interrupt generation on counter overflow.¹³ The Alpha 21264 was also the first processor to support ProfileMe, a novel profiling technique, discussed in the "Alpha's ProfileMe approach" sidebar.¹⁴

Limitations

The performance-monitoring support for these processors all suffer from a common set of problems.

Too few counters. First, the processors have only a limited number of counters (ranging from two to six) that can count concurrently. Therefore, collecting a variety of performance data requires running the application many times, because the hardware can only concurrently count a small subset of events. Even the Motorola PowerPC 7450 with six performance counters requires many application runs to collect a complete set of data for all performance events.

For many events, users might want to calculate the ratio of event count to the number of instructions retired. This ratio helps gauge a performance problem's severity, and the instruction count can confirm the consistency between different runs of the same application. So, it's often the case that one of the few counters available is used repeatedly to collect instruction counts. In general, having more counters would reduce the number of application runs needed to collect performance data.

Speculative counts. Second, for processors that detect performance events for instructions that do not complete, the problem of speculative versus nonspeculative counts arises. Most high-performance processors have deep

Alpha's ProfileMe approach

Digital Equipment Corp. (now Hewlett-Packard) developed the ProfileMe¹ method for instruction-level profiling as part of the Digital Continuous Profiling Infrastructure (<http://www.tru64unix.compaq.com/dcp/index.html>). ProfileMe differs from the profiling support offered by other processors described in this article. Instead of capturing the program counter's value upon the overflow of a performance counter, the ProfileMe approach uses hardware that, when triggered, causes the processor to collect performance event information for a single instruction as it flows through the processor pipeline. Thus, this approach collects the entire performance lifetime for one execution of the instruction. Internal registers hold this information, which includes data such as the program counter, pipeline stage latencies, cache hit/miss information (including addresses), branch target addresses, and whether the instruction retires or is cancelled. Once the instruction completes, the ProfileMe hardware generates an interrupt, and the interrupt service routine collects the information as a sample for building a performance profile. The first implementation of ProfileMe appears in the Alpha 21264/EV67 processor.²

Although collecting independent samples of instruction lifetimes can reveal much about an instruction's typical performance, on more advanced processors, the interaction of multiple instructions executing concurrently can often reveal more information about the stalls and bottlenecks that an application encounters. To better depict the interaction of concurrently executing instructions in the processor's pipeline, ProfileMe also supports paired sampling, where, for each instruction profiled, a second instruction that may execute concurrently with the first instruction is also profiled. These paired samples enable the creation of various concurrency and utilization metrics, such as an estimate of the number of cycles during a particular instruction's lifetime in which the processor did not issue other instructions.

ProfileMe's more detailed picture of how instructions flow through the pipeline permits a wider variety of metrics than is possible with more traditional profiling approaches. This information can help find and analyze performance bottlenecks to improve application performance as well as aid in the design of future processors.

References

1. J. Dean et al., "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Symp. Microarchitecture (Micro-30)*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 292-302.
2. *Alpha 21264/EV67 Microprocessor Hardware Reference Manual*, order no. DS-0028B-TE, Compaq Computer, Houston, Texas, 2000; http://www.support.compaq.com/alpha-tools/documentation/current/21264_ev67/21264ev67_hrm.pdf.

pipelines to improve performance. However, branches in a program often change the flow of instructions into the pipeline, causing a significant performance loss when the processor flushes the pipeline and fetches the correct instruction stream. To reduce the frequency of pipeline flushes, most processors use a form of branch prediction to predict the outcome of conditional branch instructions. Consequently, these processors execute most instructions speculatively (that is, before program execution determines which execution path is

correct). Some of these speculatively executed instructions are along incorrect execution paths and therefore never complete. The instructions on the correct path complete and then retire. Because speculatively executed instructions that do not complete can still cause performance events (for example, cache misses and pipeline stalls), it's often important to separate events caused by speculatively executed instructions that never retire from those that do retire. The speculative count for an event comprises events caused by all instructions, independent of whether or not the instruction that caused an event retires. Nonspeculative counts refer to event counts caused by instructions that retire.

Sampling delay. The third and probably the worst problem with most performance-monitoring hardware is the imprecision of EBS support. EBS lets the user identify the set of instructions causing the majority of occurrences of a specific performance event. Once the instruction is identified, the programmer can attempt to recode the application to avoid or reduce the performance problem's impact. Most EBS hardware simply monitors a particular performance counter and signals a performance monitor interrupt (PMI) when the counter overflows. The PMI ISR then saves the value of the program counter for the interrupted instruction. However, the current instruction is usually not the instruction that caused the performance counter to overflow.

The processor's pipeline and the delay between counter overflow and the signaling of an interrupt combine to create an arbitrary distance (measured in terms of retired instructions) between the instruction causing the performance event and the instruction whose address the PMI ISR saves. For example, consider an event-causing instruction that creates a counter overflow during the fifth stage in a 15-stage pipeline. If the event immediately caused an interrupt, at least 10 earlier instructions (assuming only one instruction per pipeline stage) remain in the pipeline. The PMI ISR would incorrectly identify an instruction prior to the event-causing instruction.

Alternatively, consider an event-causing instruction in the pipeline's last stage that makes the counter overflow. If only one instruction was in this last stage and the

processor signaled the interrupt immediately, then the PMI ISR would correctly identify the event-causing instruction. However, this situation is rarely the case. In a superscalar processor, the pipeline's last stage can contain multiple instructions. Also, several cycles elapse before the processor generates an interrupt request, allowing additional instructions to retire. In these cases, the PMI ISR would incorrectly identify an instruction that followed the event-causing instruction. Consequently, EBS typically does not correctly identify the instructions causing the performance problem. The team that developed ProfileMe found that an EBS profile of a single source of data cache misses on Intel's Pentium Pro resulted in a widely distributed profile that spanned 25 instructions.¹⁴ Inaccuracies like these make it difficult to use the EBS data and often an expert on the processor's implementation is needed to interpret the data properly.

Lack of data-address profiling. A fourth problem with most performance-monitoring hardware is that there is no support for collecting profiles of data addresses used by the program that cause performance events in the memory hierarchy (a notable exception to this is Itanium's event address register support). Just as profiles of a problematic instruction address (obtained via EBS) help identify problems in the application or operating system code, profiles of data addresses that cause performance problems (for example, cache and TLB misses) could help characterize and avoid performance problems in the memory hierarchy. To support the creation of address profiles, the processor must either capture samples of problematic data addresses or capture enough state—the instruction and the register values, for example—to compute these addresses.

The general capabilities of performance-monitoring hardware described in this article have been used extensively to analyze application, operating system, and processor performance. These analyses have helped improve not only application and operating system code but also compilers and next-generation processor designs. However, as discussed previously, the performance-monitoring support offered by most processors is limited

(too few counters, lack of support for distinguishing between speculative and nonspeculative event counts, imprecise event-based sampling, and lack of support for creating data-address profiles). The recent introduction of Intel's Pentium 4 and Pentium 4 Xeon processors provides performance-monitoring capabilities that overcome these limitations, while also providing full support for the simultaneous multithreading capabilities of the Pentium 4 Xeon processor. A companion article, *Pentium 4 Performance-Monitoring Features*, on page 72 of this issue details these performance-monitoring improvements and describes their implementation and use.

MICRO

References

1. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, order no. 245472, Intel, Santa Clara, Calif., 2002; <http://developer.intel.com/design/pentium4/manuals/>.
2. *GNU Gprof*, Free Software Foundation, Boston, 1998; <http://www.gnu.org/manual/gprof-2.9.1/gprof.html>.
3. *Intel Architecture Optimizations Manual*, order no. 242816-003, Intel, Santa Clara, Calif., 1997; <http://developer.intel.com/design/pro/manuals/>.
4. *Intel Architecture Software Optimization Reference Manual*, order no. 245127-001, Intel, Santa Clara, Calif., 1999; <http://developer.intel.com/design/PentiumIII/manuals/>.
5. D. Sima, "The Design Space of Register Renaming Techniques," *IEEE Micro*, vol. 20, no. 5, Sept./Oct. 2000, pp. 70-83.
6. T. Romer et al., "Instrumentation and Optimization of Win32/Intel Executables Using Etch," *Proc. Usenix Windows NT Workshop*, Usenix Assoc., San Diego, Calif., 1997, pp. 1-8.
7. K. Keeton et al., "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads," *Proc. 25th Int'l Symp. Computer Architecture*, CS Press, Los Alamitos, Calif., 1998, pp. 15-26.
8. *AMD Athlon Processor, x86 Code Optimization Guide*, AMD, Sunnyvale, Calif., 2002; <http://www.amd.com/products/cpg/athlon/techdocs/pdf/22007.pdf>.
9. *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*, publication no. GK21-0263-00, IBM, White Plains, N.Y., 1999; <http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256996006C28E2>.
10. *MPC7450 RISC Microprocessor Family User's Manual*, no. MPC7450UM/D, Motorola, Schaumburg, Ill., 2001; <http://www.motorola.com/brdata/PDFDB/docs/MPC7450UM.pdf>.
11. *Intel Itanium Architecture Software Developer's Manual*, rev. 1.1, vols. 1-3, Intel, Santa Clara, Calif., 2000.
12. *Ultra-SPARC-III User's Manual*, part no. 805-0087-01, Sun Microsystems, Palo Alto, Calif., 1997; <http://www.sun.com/oem/products/manuals/805-0087.pdf>.
13. *Alpha 21264 Microprocessor Hardware Reference Manual*, order no. EC-RJRZA-TE, Compaq Computer, Houston, Texas.
14. J. Dean et al., "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Symp. Microarchitecture (Micro-30)*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 292-302.

Brinkley Sprunt is an assistant professor of electrical engineering at Bucknell University. His research interests include computer performance modeling, measurement, and optimization. Sprunt has a PhD in electrical and computer engineering from Carnegie Mellon University. He previously worked at Intel where he was a member of the architecture teams for the 80960, Pentium Pro, and Pentium 4 projects. He is a member of the IEEE and ACM.

Direct questions and comments about this article to Brinkley Sprunt, Bucknell Univ., Electrical Engineering Dept., Moore Ave., Lewisburg, PA 17837; bsprunt@bucknell.edu.

For further information on this or any other computing topic, visit our Digital Library at <http://computer.org/publications/dlib>.